AN EVALUATION OF LOW-OVERHEAD PARTIAL FLOW SENSITIVITY

BY

JAMES W. PLAYER

B.S., University of Illinois at Urbana-Champaign, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Flow-sensitivity with respect to pointer analysis represents a technique that is typically classified infeasible due to scalability limitations. This thesis serves as an evaluation of partial flow-sensitivity, which is a scalable technique by which much of the benefit provided by flow-sensitivity is offered. Two flow policies beyond the initial notion of partial flow-sensitivity are proposed and evaluated in order to provide further insight into the relationships between initial complexity, scalability, and result quality. Empirical results are gathered using counting methods that extrapolate the effects of partial flow-sensitivity as though no object replication has occurred. Finally, these results are compared across the variations of flow-sensitivity and their effects upon scalability are discussed.

ACKNOWLEDGMENTS

I first thank my parents for their unending support in every respect of the word and for the foundations they helped instill in me, which made my work possible. I would also like to express my gratitude to my brothers and sister who contributed to my competitive nature and encouraged me to live well.

I would also like to thank my adviser Wen-mei Hwu for providing me with an opportunity to work in a first-rate graduate study environment. His guidance has most certainly served as an inspiration in my life and will not be forgotten.

During my graduate work at the IMPACT research group, I had the pleasure of meeting some talented members who influenced me greatly with their insight, work ethic, and patience. I take this opportunity to thank John Sias, Erik Nystrom, Ian Steiner, Sain-zee Ueng, and Hillery Hunter for their guidance and friendship.

Lastly, I thank Intel Corporation, DARPA/MARCO GSRC, the University of Illinois, and the Department of Electrical and Computer Engineering for their financial support of myself and the IMPACT research group.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Today's computer architectures are becoming increasingly complex. Not only are concerns such as branch misprediction penalties and heat dissipation placing boundaries on how far deep pipelining can enhance performance, but a recent study shows that current pipeline depths are approaching the limit of realizable performance [1]. Compounding the problem, the increasing discrepancy between processor core and memory access speeds is quickly choking off performance from modern architectures. This problem in particular has motivated computer architects to search for answers in areas such as compiler-hardware interactions.

In order to facilitate various compiler transformations and improve performance in a meaningful way, the compiler must be able to make intelligent decisions while transforming code. In order to make an impact on the memory latency problem, the compiler must be provided with precise knowledge about how a program accesses memory. That is, if the compiler asks whether two memory accesses overlap in memory, there should be a tool that can provide an accurate response. Pointer analysis is the name given to a compiler tool that does just that.

Traditional pointer analysis typically can be divided into two kinds. *Andersen*[1] style [2] assignment modeling tracks objects in a program seperately from each other. The analysis results yielded by an Andersen style analysis are regarded as highly accurate, but this accuracy

[1]Also know as *subtyping* or *inclusion* based assignment modeling.

Table 1.1 Pointer analysis terms. Andersen and Steensgard are two options for modeling assignment behavior. The remaining terms are features that may be included or excluded from a pointer analysis.

Term:	**Description:**
Andersen Modeling	inclusion based assignment modeling; tracks objects separately from each other
Steensgard Modeling	equivalence based assignment modeling; combines objects that are found to be referenced by the same memory location
Context Sensitivity	interprocedural analysis that tracks pointer behavior independently for different call paths of a procedure
Heap Specialization	typically coupled with *context sensitivity* to differentiate objects dynamically allocated along different call paths by the same static call
Field Sensitivity	distinguishes the fields of an object from one another
Flow Sensitivity	accounts for program control flow when modeling assignments

comes at a price of limited scalability. The other kind of pointer analysis is called *Steensgard*[2] style [3]. In contrast with Andersen style analysis, Steensgard analysis merges all objects found to be referenced by a single location in memory and tracks them as a single object. The result of this method for assignment modeling is a class of pointer analyses that are efficient and scalable but significantly less accurate than their inclusion based counterparts. The assignment modeling is not the only aspect of pointer analysis that affects the precision and scalability. Other features are typically options that may be turned on/off. Such features include *context sensitivity* [4], *heap specialization* [5, 6], *field sensitivity* and *flow sensitivity*. Table 1.1 provides brief descriptions of these features.

[2] Also known as *unification* or *equivalence* based assignment modeling.

Given the numerous choices and features available for a pointer analysis implementation, the decision of precision versus scalability becomes difficult to balance. Nystrom [7] developed algorithms within a framework called Fulcra that sought to combine novel approaches to Andersen's style, context, and field sensitivity with techniques to automatically limit problem size growth to help strike a balance between precision and analysis time.

Fulcra does not offer any form of flow sensitivity as presented in [7] but suggests two options for the addition of such a feature. One option is to add full flow sensitivity [8, 9] to the core of Fulcra. However, previous work [10, 11] suggests that flow-sensitive analysis algorithms are too expensive to run on large programs. This is a direct contradiction to the objective of offering precision without sacrificing scalability. A second option mentioned by [7] is a notion of partial flow sensitivity [12] which conditions the input to Fulcra with static single assignment (SSA) [13] and enables Fulcra to process renamed variables in a flow-insensitive manner. This option seems more apt to balancing scalability with an increase in precision.

Given this notion of partial flow sensitivity and the fact that flow-sensitive results tend to be more precise than flow-insensitive ones, a question arises: how much flow must be represented in order to make a meaningful difference in the precision of the result? Naturally, renaming variable with SSA creates more work for the pointer analysis. Hence, if a small amount of renaming gains a large amount of precision, partial flow sensitivity becomes an extremely attractive option. Similarly, if the pointer analysis scales well with a large amount of renaming, then this method is enticing because of the low overhead of producing SSA.

This thesis describes the principles and tasks involved in adding partial flow sensitivity to the Fulcra framework and empirically evaluates this feature at varying levels of renaming. The aim of this work is to classify the amount of flow-sensitivity that gives the greatest precision benefit while sacrificing as little scalability as possible. Chapter 2 provides further background on the concepts that build this thesis. Supplementary details on pointer analysis as well as flow-sensitivity are covered. SSA is discussed along with some features that simplify and enhance the SSA representation. The chapter concludes with a brief description of the IMPACT and Fulcra frameworks. Chapter 3 presents implementation details for the various pieces of this work. SSA features implemented in the framework are addressed as well as the limitations of variable coverage by SSA. Next, the flow-sensitivity policies are described along with their roll in the evalutation of this work. Lastly, the specific modifications to the Fulcra framework are discussed. Chapter 4 displays the results from various evaluations of this notion of partial flow sensitivity. Chapter 5 suggests some furture work that may improve upon these results. Finally, Chapter 6 offers some conclusions and closing remarks on this work.

CHAPTER 2

BACKGROUND

This chapter covers topics introduced by Chapter 1 in greater depth. SSA has been mentioned as a tool this work hinges off, and appropriately, Section 2.1 examines basic SSA concepts and features that are relevant to this work. Improving pointer analysis in a meaningful way is the goal of this thesis. Therefore, a discussion of a way to interpret an internal pointer analysis representation is provided by Section 2.2. Finally, Section 2.3 presents some background information about the compiler and pointer analysis framework used to evaluate partial flow sensitivity in this thesis.

2.1 Static Single Assignment

SSA [13] is a style of internal compiler representation that annotates variable names with subscripts such that only a single definition exists for each subscripted variable name. The typical goal of performing SSA analysis is to provide a regularity to the structure of program data flow thereby simplifying subsequent analyses.

This section highlights various aspects of SSA that are important to accomplishing a partial flow sensitive implementation. Section 2.1.1 discusses ϕ-nodes because an intuitive understanding of these operations facilitates a later discussion of what to do with them in a flow-insensitive pointer analysis framework. Sections 2.1.2 and 2.1.3 elaborate upon two optimizations that eliminate meaningless ϕ-nodes from the SSA representation. Such ϕ-nodes convey no information, and therefore, their presence or absence does not affect pointer analysis results. They are in fact undesirable because they introduce unnecessary relations into the pointer analysis and potentially slow it down. Finally, Section 2.1.4 mentions μ-nodes as defined by gated SSA representations because one of the flow policies makes use of these nodes.

2.1.1 ϕ-nodes

The ϕ-nodes present in an SSA representation are simply a way to compensate for different SSA-subscripted variable names being live in the predecessors of a merge in program control flow. The ϕ-node performs an imaginary operation that merges these different definition names into a single new name. In general, a ϕ-node has two or more operands corresponding to various SSA subscript versions of a single variable and defines a new version of this variable. The operands of a ϕ-node need not all be uniquely subscripted, as a single definition may reach a merge in control flow through different paths. On the other hand, a ϕ-node containing all operands of the same subscript serves no purpose because the effect of merging a single definition with itself is trivial.

Figure 2.1 provides an example of placement and use of a ϕ-node. Figure 2.1(a) depicts a merge in control flow where x contains different values, depending on which predecessor precedes block (3) in the dynamic instruction stream. In order to reconcile this ambiguity, Figure 2.1(b) shows that x is given a unique name for each definition that occurs in the predecessors

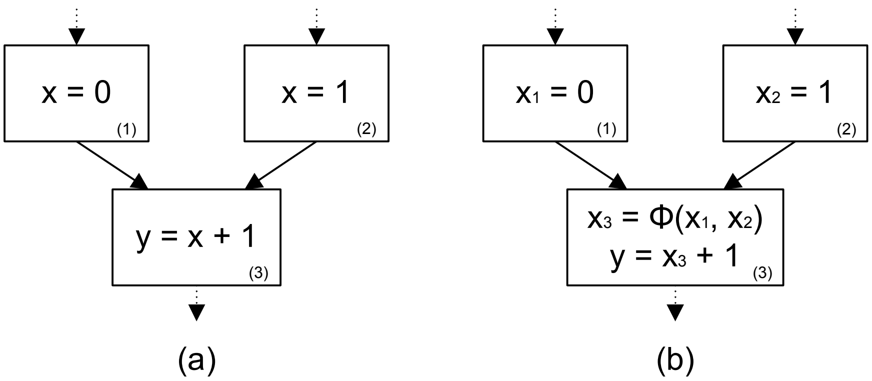

Figure 2.1 ϕ-node example. (a) A simple merge in control flow where the use of x in block (3) has different values live in from blocks (1) and (2). (b) SSA is computed for x and a ϕ-node is placed at the top of block (3).

of block (3) and a ϕ-node is placed at the top of (3) to merges the definitions of x. This ϕ-node defines a third version of x and is ultimately used in a subsequent computation in (3). In the presence of the ϕ-node, x_3 is clearly shown to be either the value of x_1 or x_2.

2.1.2 Minimal SSA

The name "minimal SSA" refers to the minimization of the number of ϕ-nodes added to an SSA representation. The brute-force method for placing ϕ-nodes would be to add one for every SSA variable at the start of every basic block with more than a single predecessor. Minimal SSA ensures that ϕ-nodes are only placed at merge points that actually serve a purpose for the SSA variable being considered.

The dominance frontier [13] is the tool that enables minimal SSA. The dominance frontier of any queried point in a control flow graph is a collection of edges that begin at a point dominated by the queried point and end at a point that is not strictly dominated by the queried

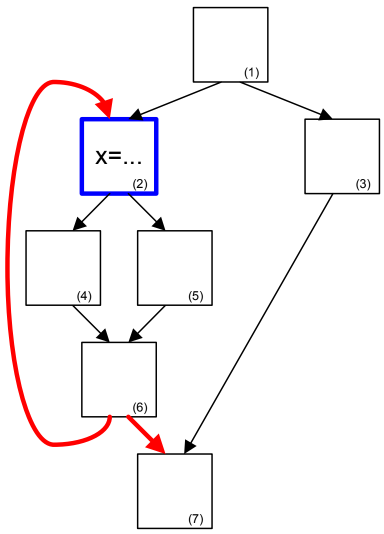

Figure 2.2 Part of a control flow graph. The dominance frontier of block (2) is shown by the bold arcs.

point. In a sense, the dominance frontier specifies the boundaries of dominance for any given point in a control flow graph.

The dominance frontier is important to minimal SSA because it shows precisely where ϕ-nodes need to be placed for every assignment in the procedure. The procedure of placing ϕ-nodes is to simply traverse the list of assignments in a procedure and create a ϕ-node at every merge point on that assignment's dominance frontier. If a merge point lies inside of an assignment's dominance frontier, then it does not require a ϕ-node corresponding to that assignment. This is because the definition in question is live-in from all predecessors of the merge point. Similarly, if a merge point lies outside of an assignment's dominance frontier, it does not need a corresponding ϕ-node because the placement of ϕ-node ensures the SSA variable is not live outside of its dominance frontier.

Figure 2.2 depicts a piece of a control flow graph. Given this bit of control flow, the dominance frontier for basic block (2) is denoted by the thick edges in the figure. Suppose ϕ-nodes are being placed for an assignment to x that lies within block (2). Given the dominance frontier for block (2), ϕ-nodes for x will be placed at the start of blocks (2) and (7). Notice that no ϕ-node is placed for x at the top of block (6) even though it is a merge point in the control flow graph. A merge definition is not required in block (6) because the x value defined in (2) will be live on all paths into (6) given that x is not redefined in between. Fortunately, this assumption is valid because SSA only permits a single assignment to any given subscripted variable name. Therefore, by placing ϕ-nodes only on the dominance frontiers of all definitions, the number of ϕ-nodes created is less than the number created by the brute-force method.

2.1.3 Pruned SSA

Often times, minimal SSA does not truly produce the minimal number of ϕ-nodes in a procedure as its name suggests. Generating pruned SSA [14] involves a postprocessing pass over the code to determine which ϕ-nodes are actually consumed by uses of the variable version it defines. If a ϕ-node defines a version that is never consumed, it can safely be deleted from the SSA representation. After the removal of a dead ϕ-node, the operands to that ϕ-node must be evaluated to see if the dead ϕ-node was the only use. Using the dominance frontier to place ϕ-nodes for minimal SSA considers only the definitions of variables in an attempt to reduce the number of ϕ-nodes added to an SSA representation. Pruning an SSA representation, on the other hand, uses the use information to ϕ-nodes which are meaningless.

2.1.4 μ-node classification

Gated SSA [15] defines μ-nodes as two-input definition merging operation with an additional predicate input. It is basically a special case of a ϕ-node. The μ-node is placed at the top of a loop and selects a value coming in from above the loop or from the previous iteration of the loop based on the predicate value. The work for this thesis does not rely on the behavior of a μ-node as defined in a formal gated SSA implementation. In other words, we do not care about breaking a ϕ-node down into two-input components, nor do we care to track the origins of the operands to a ϕ-node based on the predicate value. The only relevant characteristic of the μ-node is the fact that it is derived from a ϕ-node placed at the top of a loop. This μ classification shall be revisited in Chapter 3 when the implementation of IMPACT's SSA library and the Loop Merge flow policy are discussed.

2.2 Pointer Analysis

The main benefactors of ideas presented in this thesis manifest themselves in pointer analysis. As such, some peripheral knowledge on this subject is required to motivate this work. This section provides overviews of two topics that help explain and drive implementation considerations. Section 2.2.1 describes the grammar and solution rules of a pointer analysis constraint graph, and Section 2.2.2 expounds upon partial flow sensitivity.

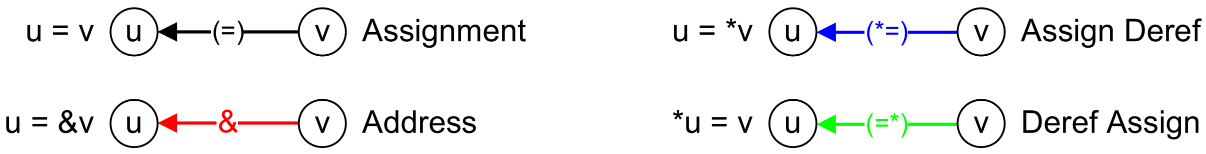

Figure 2.3 Constraint graph grammar where & specifies the address operator and * is the dereference operator. These four relations are sufficient to describe all assignment relations in the absence of field sensitivity.

2.2.1 Constraint graphs

The constraint graph is a representation of the assignment modeling that occurs in pointer analysis. Constraint graphs throughout this thesis use a grammar [16] similar to Fulcra. Fulcra's constraint graphs consist of five edge types that represent the basic object relations and nodes that signify objects in the program or a temporary object (necessary to represent a complex relation not covered by a single edge such as $**p$). In order to simplify matters in the context of this thesis, one of the edges present in Fulcra's constraint graphs (the *skew* edge) is omitted because its use only pertains to field sensitivity, a topic not relevant to partial flow sensitivity. The edge types described in Figure 2.3 define a complete enough grammar for all the examples discussed in this thesis. The direction of an edge in a constraint graph represents the flow of value in the corresponding assignment. So for a simple assignment $u = v$, the edge modeling this assignment will start at the v node and point to the u node because the value of v is flowing into u.

Once an initial constraint graph is constructed additional relations can be derived from the initial set. For example, if an assignment edge occurs in the constraint graph, the node at the head of that arc effectively receives all the values that flow into the node at the tail.

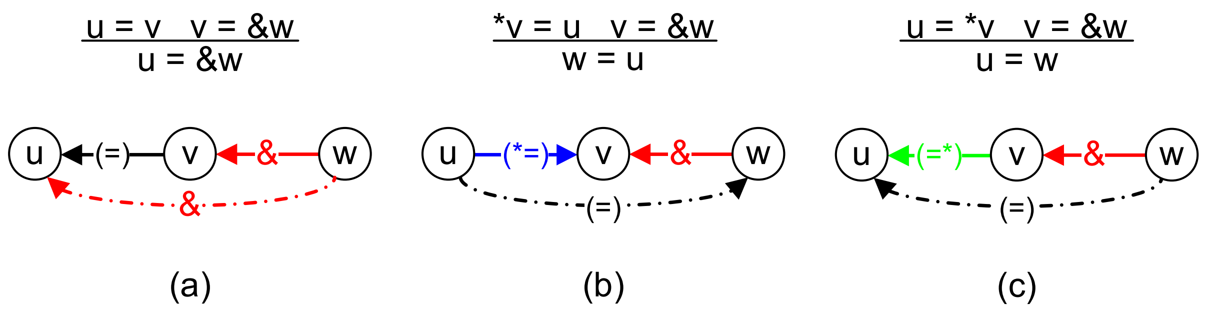

Figure 2.4 The three rules to solve a constraint graph. For the equations, if both conditions in the numerator are satisfied, then the derivation in the denominator is true as well. In the graphical representations, if the relations denoted by the solid edges exist, then the dashed edge may be added to the graph. (a) Propagation of an $(\&)$ edge across an $(=)$ edge. (b) Addition of an $(=)$ edge from u to w when v may point to w and the value of u is deref-assigned into v. (c) Addition of an $(=)$ edge from w to u when v may point to w and the value pointed to by v is assigned into u.

Hence, any addresses that may be stored in the tail node, may be stored in the head node as well. This is a practical description of the constraint graph derivation rule from Figure 2.4(a). There are two other derivation rules capable of adding assignment edges to the graph in the presence of address and deref-equals edges (Figure 2.4(b)) or address and equals-deref edges (Figure 2.4(c)). A constraint graph is not considered solved until these three identities have been applied exhaustively to the initial constraint graph. This is referred to as performing a transitive closure on the constraint graph.

2.2.2 Partial flow sensitivity

Flow sensitivity is a fairly intuitive concept. It dictates that the control flow of a program be considered when modelling assignment behavior during pointer analysis. Conversely, a flow-insensitive analysis means that assignments in a program section, typically a procedure, are

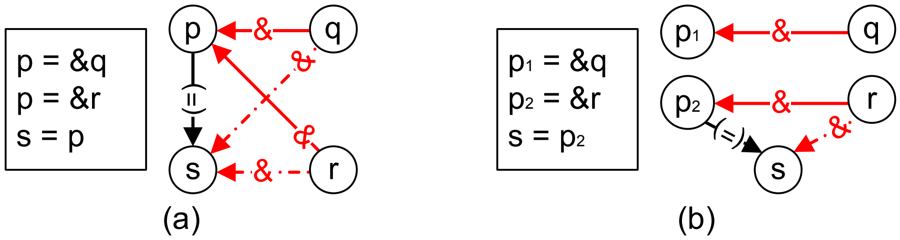

Figure 2.5 Flow sensitivity example with positive influence on precision. Solid lines in the constraint graphs indicate initial edges and dashed lines imply edges added by applying rules from Figure 2.4. (a) Sequential assignments of a pointer p are not differentiated in the absence of partial flow-sensitivity. (b) Partial flow-sensitivity is applied and now sequential assignments apply apply to unique objects.

regarded as though they may occur in any order. An analysis with this simplification imposed upon it will return results that are imprecise in varying degrees. The partial flow-sensitive policies proposed and evaluated by this thesis offer an alternative that can be made both precise and scalable, given the flow-insensitive framework it is built upon.

Here is an example of how partial flow-sensitivity can improve precision by renaming variables using SSA. Consider the C code excerpt and corresponding constraint graph relation in Figure 2.5(a). This is the result of a flow insensitive analysis. The addresses of q and r are both assigned into p and the value of p is assigned into s. The analysis propagates the constraint relations across the $s \leftarrow p$ assignment edge using the rule from Figure 2.4(a). The final constraint graph shows that s may point to q. A quick inspection of the code in Figure 2.5(a) reveals that this is untrue given the order of assignments in this code. When s is assigned the value in p, the only possible value p may contain is the address of r.

Now examine the code and constraint graph in Figure 2.5(b). The code in Figure 2.5(b) is identical to Figure 2.5(a) except the variable p has been annotated with SSA subscripts. When

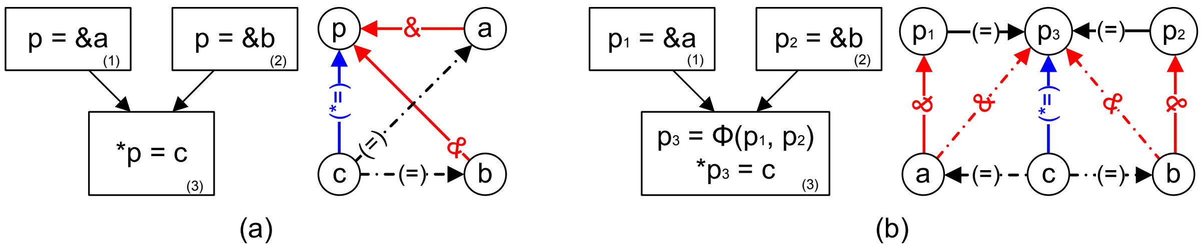

Figure 2.6 Example of flow sensitivity not contributing to precision. Solid lines in the constraint graphs indicate initial edges and dashed lines imply edges added by applying rules from Figure 2.4. (a) Control flow merge implies both definitions of p may be pointed to in block (3). (b) Partial-flow sensitivity is applied to p, but the analysis conclusions are identical to (a).

a constraint graph is formed and solved in a flow insensitive manner, the result is different due to the variable renaming that has been imposed. The constraint graph in Figure 2.5(b) demonstrates an improvement over the one in Figure 2.5(a) because the false address edge between q and s is omitted.

Although the addition of flow is capable of eliminating false relations from the pointer analysis internal representation, as in Figure 2.5, some situations may not benefit from the addition of partial flow sensitivity. Figure 2.6 is an example of a circumstance in which flow contributes nothing to the precision of the analysis. Note that the ϕ-node is treated as if it were a series of assignments from each operand to the LHS variable when the constraint graph is derived from SSA. Figure 2.6(a) represents an initial code example and constraint graph solved in a flow-insensitive manner. The addresses of both a and b are stored to p. The value of c is then stored to the address in p in a subsequent assignment. The solved constraint graph indicates that the value of c may be assigned into both a and b depending on the control flow path taken by the program at run time.

Figure 2.6(b) shows the same code example as Figure 2.6(a) except p has been renamed and a ϕ-node has been added to the top of block (3) to comply with SSA specification. The corresponding constraint graph solution is given and shows a separation between the relations of a, b and c across different versions of p. However, relations between a, b and c become identical to Figure 2.6(a) after the constraint graph is solved. The constraint graph again shows that c may be assigned into a or b depending on execution path.

In this example, two nodes and four edges are added to the constraint graph when partial flow sensitivity is applied, but the overall relations between variables are unchanged. Therefore, complexity is introduced into the analysis, and compute time is sacrificed with no precision increase in exchange. However, not every merge point in a control flow graph spells disaster for partial flow sensitivity. In fact, the example in Figure 2.6 need only be altered slightly to become a case that benefits from partial flow sensitivity.

Figure 2.7 is a modified version of the example in Figure 2.6. The difference between these examples is that Figure 2.7 contains an additional assignment in block (2). This assignment stores the value of d to the memory address pointed to by p. When a constraint graph is computed and solved with the flow insensitive approach in Figure 2.7(a), the value of d appears as if it may be stored in either a or b. However, inspection of the control flow in this example reveals that d can only be stored to b because the store occurs immediately after p is defined by the address of b.

Figure 2.7(b) shows an SSA representation of p and the constraint graph that results from this transformation. In this revision, notice that d is now stored to p_2. The address of b is the

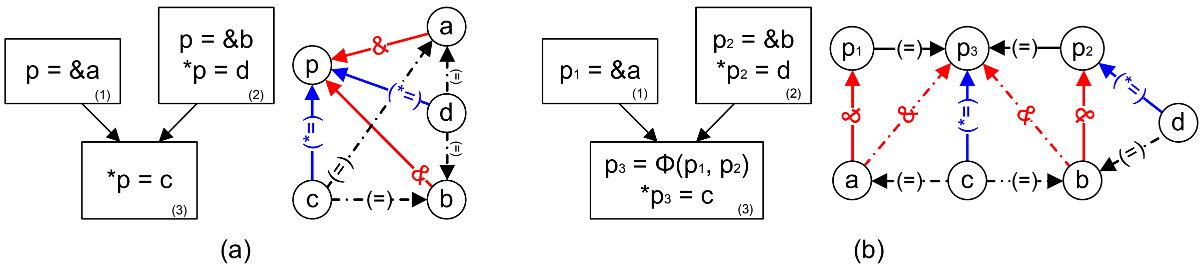

Figure 2.7 Modified version of Figure 2.6 in which flow sensitivity improves precision. Solid lines in the constraint graphs indicate initial edges and dashed lines imply edges added by applying rules from Figure 2.4. (a) In the absence of partial flow-sensitivity, d's appears to propagate into a and b. (b) Partial flow-sensitivity is applied to p, revealing that d's value only propagates into b.

only value stored to p_2 because it is an SSA renamed variable version. The resulting partial flow sensitive constraint graph reflects the ordering of assignments surrounding d because its value of d no longer appears as if it may be assigned into a. This decoupling of d from a comes at a price of two additional nodes and four new edges in the constraint graph, but the precision of the analysis result benefits from these costs.

Examples in this subsection thus far have demonstrated how partial flow sensitivity may or may not contribute to result precision. Complexity introduced into the constraint graphs in these examples has taken the form of extra nodes and potentially redundant edges. However, complexity can also manifest in a structural manner in the constraint graph. Specifically, a cycle of assignment arcs in the constraint graph yields a situation in which pointer analysis results for each node in the cycle are identical. As an optimization, Fulcra performs a cycle-detection pass on all constraint graphs to identify these cycles and merges all these nodes together. Therefore, any cycles of assignment edges introduced into the constraint graph by

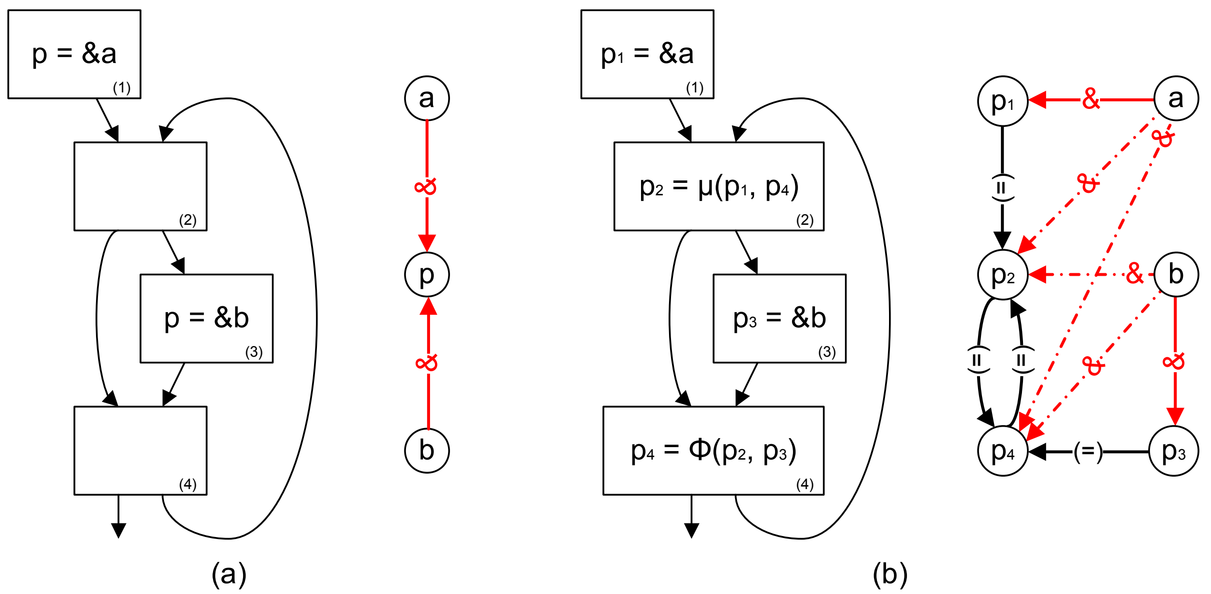

Figure 2.8 Flow sensitivity example that introduces a constraint graph cycle. Solid lines in the constraint graphs indicate initial edges and dashed lines imply edges added by applying rules from Figure 2.4. (a) Control flow loop with a definition of p above and within the loop. (b) Partial flow-sensitivity is applied to p and creates an assignment edge cycle in the constraint graph.

partial flow sensitivity do not contribute to analysis results in a positive way and actually create more work for Fulcra as it identifies and eliminates them.

In light of the analysis time penalty caused by assignment edge cycles, Figure 2.8 establishes a simple case in which partial flow sensitivity introduces such a cycle. In Figure 2.8(a), a pointer, p, is defined above the entrance to a loop. The loop contains some internal control flow, and p is redefined on one of the paths through the loop. The relations derived from flow-insensitive analysis are remarkably simple. In effect, p may point to a or b.

Figure 2.8(b) reveals the effects of computing SSA on p and applying these changes to the constraint graph. In the SSA version of the code, two ϕ-nodes are added. One occurs at the top of the loop in block (2). This ϕ-node merges the definition from outside the loop with the definition that arrives from the loop's back-edge. The other ϕ-node is a result of control flow within the loop and is placed at the bottom in block (4). The constraint graph shows the assignment edge cycle involving nodes p_2 and p_4. Clearly, this cycle is a direct result of partial flow sensitivity because its components are subscripted versions of the same variable. The occurrence of the cycle can be attributed to a definition of p lying within the loop and the merge definition from the top of the loop (p_2) reaching the bottom of the loop. Indeed, the members of the cycle have identical relations to both a and b, and therefore, do not contribute to the precision of the result.

In summary of this subsection, the examples presented have served to highlight a variety of situations in which flow sensitivity may contribute in a positive or negative manner. Figure 2.5 is an example of straight-line code, which lends itself well to flow-sensitive analysis. Figures 2.6 and 2.7 are two similar situations involving a merge in control flow. The former is a situation of such simplicity that the notion of flow added nothing meaningful to the constraint graph. The latter, on the other hand, added a dereference assignment to one of the predecessor blocks to the merge and demonstrated partial flow-sensitivity's ability to isolate the assignment relations of the predecessors to a merge. Finally, Figure 2.8 provides an example of how a loop in control flow may yield an assignment cycle in the constraint graph when partial flow-sensitivity is introduced. None of these examples demonstrate a situation that is harmful to the quality of the analysis result itself. Rather, the usefulness of flow sensitivity is categorized by whether it helps to eliminate false dependences from the constraint graph derivations while adding as little complexity as possible.

2.3 Platform

This thesis evaluates the addition of partial flow sensitivity, and this evaluation requires a compiler and pointer analysis infrastructure to build upon. This section provides overviews of the IMPACT research compiler, Section 2.3.1, and Fulcra, the specific pointer analysis module built for IMPACT, Section 2.3.2.

2.3.1 IMPACT

IMPACT is an acronym for *Illinois Microarchitecture Project Utilizing Advanced Compiler Technology*. As the acronym partially suggests, it is a research compiler that has been developed by students and faculty at the University of Illinois at Urbana-Champaign under the guidance of Wen-mei Hwu since the early 1990s. As a research compiler, one of its primary concerns is to provide a flexible framework capable of performing powerful optimizations for many target architectures. Target architectures over the years have included various incarnations of EPIC machines, both simulation and real-world targets. Although target flexibility is a first-order design parameter of the IMPACT compiler, source code flexibility has always been limited to C. C++ support has been an ongoing goal for expanding source code flexibility, but it is not fully supported at present.

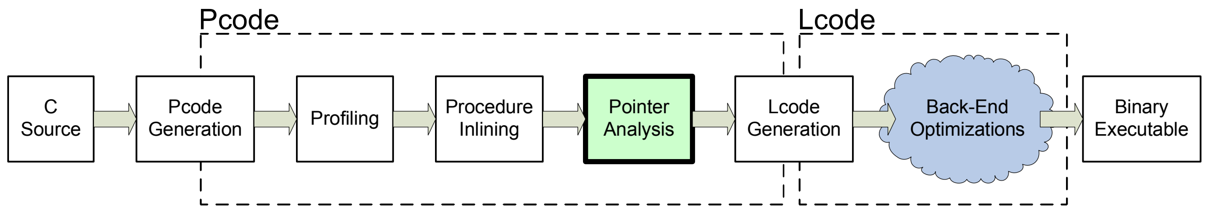

Figure 2.9 IMPACT's front-end path.

One of the keys giving IMPACT the flexibility it enjoys is the careful selection and implementation of its intermediate representations. Figure 2.9 depicts the general flow of compilation process in IMPACT. C source code is read in and converted to IMPACT's general abstract syntax tree notation, *Pcode*, in the Pcode generation module. During Pcode generation, a typical AST representation is formed with complex expressions. This is additionally responsible for *flattening* the Pcode, which implies simplification of complex expressions by breaking them into smaller components that store to internal temporaries. Next, profiling information is annotated to the Pcode during a profiling phase. Then, profiling data is consumed during procedure inlining. The information is used to decide which call sites are appropriate candidates for inlining. After inlining is complete, the compiler enters the pointer analysis stage. The pointer analysis module, Fulcra, is the subject of Nystrom's [7] Ph.D. dissertation as well as the platform this thesis uses to evaluate the addition of partial flow sensitivity. The basic function of Fulcra is to supply the subsequent low-level code generation phase, *Lcode*, with information regarding which loads and stores in the program may refer the same memory locations. This concludes the use of the front-end representation, Pcode, in IMPACT as the Lcode generation phase converts Pcode into a low-level 3-address code representation. Lcode

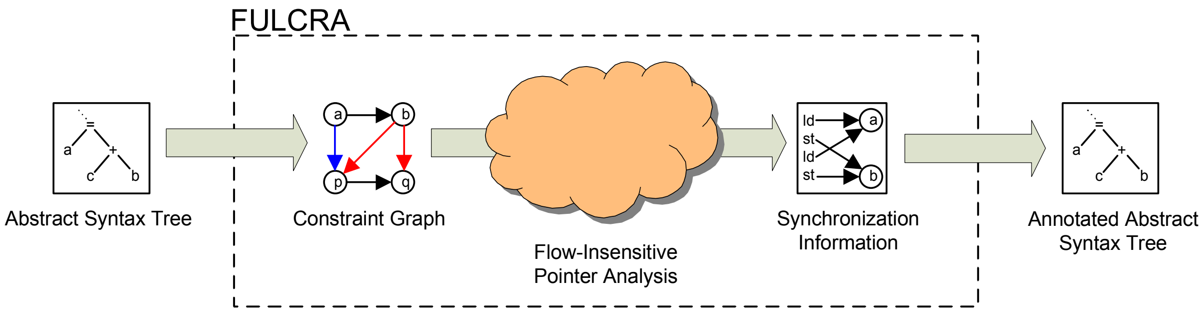

Figure 2.10 Fulcra's original path of representations.

is then passed about the back-end of IMPACT as various instruction-level parallelism optimizations are performed. Eventually, the optimization process concludes, and a binary executable is produced.

2.3.2 Fulcra

As stated in Section 2.3.1, Fulcra is the pointer analysis module used by and developed for the IMPACT compiler platform. The general stages of Fulcra's internal representations are represented in Figure 2.10. IMPACT's front-end abstract syntax tree representation is read in a procedure at a time by Fulcra, and initial constraint graphs are formed. Once the initial constraint graphs are created, Fulcra performs a complex series of derivations upon them, taking call-path, object field and heap specialization into consideration. The final constraint graphs contain summaries of the constraint graphs from callee functions, and therefore, fully derived assignment interactions between a procedure's variables and its callees. After the final constraint graphs are available, Fulcra performs an annotation pass over the initial Pcode during which it consults the appropriate constraint graph to determine which memory objects each load/store in the program may access.

The flexibility of the Fulcra framework is evident in the fact that it only touches IMPACT's Pcode representation in two places. Any changes to the internal compiler representation that feeds Fulcra will merely require revisions to Fulcra's initial constraint graph construction routines and the synchronization generation phase. That is, unless a change to the compiler's intermediate representation implies a policy change within Fulcra (as the introduction of subscripts by partial flow sensitivity does).

CHAPTER 3

IMPLEMENTATION

This chapter provides an overview of various implementation details relevant to achieving a partial flow sensitive analysis on top of a flow-insensitive pointer analysis framework. SSA is used extensively to achieve partial flow sensitivity. In light of this fact, some specifics regarding IMPACT's SSA library are discussed in Section 3.1. Next, Section 3.2 touches on the specific policies used to vary the amount of flow exposed by partial flow sensitivity. Finally, a quick description of modifications made to the specific pointer analysis framework is given in Section 3.3.

3.1 Static Single Assignment

IMPACT's SSA library is a minimal, pruned SSA implementation with support for μ-node identification. The SSA computation process begins with IMPACT's abstract syntax tree (AST) internal representation. The AST is flattened, which transforms complex expressions into multiple simpler ones. This flattened AST resembles a three-address representation in many respects because of expression simplification, but basic C characteristics are preserved due to the overall statement structure of the AST. Next, a control flow graph (CFG) is derived from the flattened AST. Finally SSA is computed on top of the CFG.

3.1.1 Variable classes

The only variable class supported by the IMPACT SSA library is local variables that do not have their address taken. Global variables are not supported because their values may be modified or used during the execution of a function that is called by the procedure being analyzed. These characteristics of global variables imply definitions and uses that are not made explicit by the expressions contained in a particular function. Adding support for global variables in SSA entails one of two options. A conservative approach can be taken, treating every function call as a definition and use of every global variable observed in a given procedure, or an interprocedural analysis can be performed which searches all children in the call graph for definitions and uses of global variables relevant to the a given procedure. The conservative approach is typically regarded as infeasible because of the inevitable explosion in the size of the SSA representation, and dividing a global variable into many, potentially false, versions may have an adverse effect on an optimization that consumes the SSA. Interprocedural searches through call graph children are expensive operations, and would undoubtedly make SSA computation a highly invasive process to the rest of the compiler.

Restricting address-taken local variables is done for a reason similar to global variables. That is, if a local variable has its address taken at some point in a procedure, the value of its address may be passed in some convoluted way to a callee procedure that may use and modify the local variable object by dereferencing its address. Again, the basic problem is that this class of variables may be used or modified by the call-graph children of a given procedure. A simple solution to this problem is to perform an *escape analysis* on any local variable that has its address taken to identify which address-taken local variables actually demonstrate this problem. However, the coverage of local variables is typically high enough in the face of the address-taken restriction that escape analysis is deemed unnecessary and results presented in Section 4.1 support this conclusion.

3.1.2 Def-use chains

Inside the CFG, there are various definitions and uses of variables from the original program and temporaries created in the flattening process. The SSA module begins by collecting and tracking all the local variables in a particular procedure. A structure is annotated to all definitions of local variables throughout a procedure and serves as the *def* structure for the *def-use chains*. As ϕ-nodes are placed within the procedure, they are also allotted a def structure. After SSA subscripts are computed throughout the procedure, it becomes obvious which variable definition reaches a particular variable use by matching subscripts. Hence, variable uses are given a pointer that refers to the def structure annotated to the defining expression. In addition, a use list is maintained within the def structure. Naturally, members of the use list point to variable uses of the particular def that lists them. Finally, all def structures are linked together in a linked list fashion. This def-use structure arrangement allows the user to access both definition and use information for any version of a particular variable regardless of which actual variable instance he starts from.

3.1.3 μ-node classification

In the IMPACT SSA library, ϕ-nodes are classified as μ-nodes when they are found to exist in the header block of a loop. Given the simplified definition of a μ-node stated in Section 2.1.4, the identification method becomes quite simple. If any of the inputs to the ϕ-node arrive from the back-edge of a loop, then it is a μ-node. The definition of a back-egde dictates it begins from a basic block that is dominated by the block where it terminates. Given this definition, the μ-node detection procedure examines each operand of a given ϕ-node. It then locates the block which contains the definition of the operand in question (by using the def information embedded in the use) and checks if the ϕ-node's block dominates the block containing the definition of the operand. If the dominance query gives an affirmative response, then the ϕ-node is marked as a μ-node.

3.2 Flow-Sensitivity Policies

Using SSA to enforce a notion of flow upon a flow-insensitive pointer analysis framework introduces matters that are potentially detrimental to the analysis time of a program. The goal of this thesis project is to evaluate the quality of the pointer analysis result with varying degrees of flow imposed upon the input program and weigh the execution time penalty against any improvement in the solution. To that end, this section discusses the methods used to attain varying degrees of flow from an input benchmark. Specifically, it will describe the subscript-merging policies now available in the IMPACT SSA library. The hope of merging subscripts is to reduce the number of situations that are likely hazardous to Fulcra's compute-time by

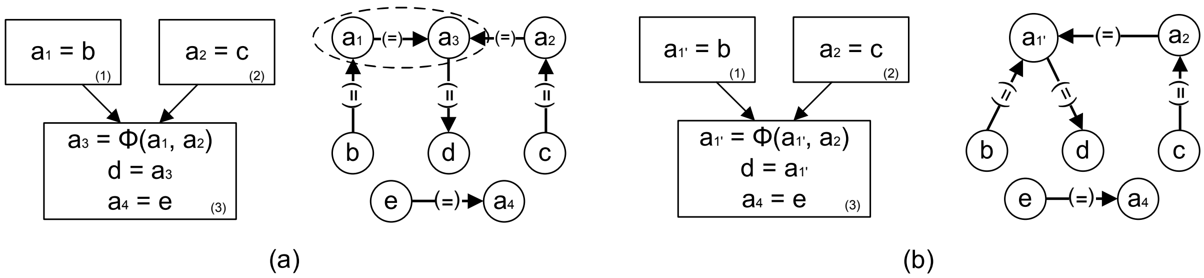

Figure 3.1 Merging operation example. (a) Subscripted variables a_1 and a_3 are targeted for a merge operation. (b) Subscripted variables a_1 and a_3 have been merged and are now called $a_{1'}$.

weeding out potentially unnecessary flow. Potentially unnecessary flow situations targeted by merging policies include cycles in the constraint graphs as well as the sheer number of nodes represented in the constraint graphs.

A merge policy is a heuristic that attempts to reduce the complexity introduced to the initial constraint graphs by performing a merging operation on various SSA subscripted definitions of a single variable. The effect of a merging operation on the constraint graph is to combine the nodes represented by the merged SSA definitions into a super-node with all the inbound and outbound edges that occurred in the original nodes. Figure 3.1 demonstrates the effects of a merging operation upon the SSA representation and constraint graph. Figure 3.1(a) shows an initial state of the SSA and constraint graph. Variables a_1 and a_3 are targeted for merging, and their corresponding nodes are circled in the constraint graph. In Figure 3.1(b), the merging operation has taken place, resulting in a new variable name, $a_{1'}$. Notice the ϕ-definition in block (3) defines $a_{1'}$ and carries it as an operand. Also observe that $a_{1'}$ is now defined in

Table 3.1 Flow sensitivity policies. Policies are listed in order from most aggressive to least in terms of the amount of flow introduced to Fulcra.

Policy:	Abreviation:	Description:
Full SSA	full	Full, unmodified SSA is fed into Fulcra.
Loop Merge	loop	Merge all defs in a loop into a single subscript.
Disjoint Lifetime	djlt	Merge all variable versions ocurring within disjoint lifetimes.
None	none	No flow sensitivity is introduced into the Fulcra framework.

two static locations, once in block (1) and again in (3). These observations imply that once a merging operation is performed, the resulting representation is no longer valid SSA.

The modifications to Fulcra that are discussed in Section 3.3 allow it to run in its original flow-insensitive mode and three flow-sensitive modes featuring varying levels of subscript-merging. In order from most to least aggressive degrees of flow, the policies are: Full SSA (full), Loop Merge (loop), Disjoint Lifetime (djlt), and None (none). Table 3.1 provides condensed descriptions of these flow policies. The following sections describe the three flow-sensitivity policies and their intended effects on the analysis result and analysis time.

3.2.1 Full SSA

Full SSA performs no subscript-merging and, therefore, requires no extra processing after the SSA representation has been computed. The raw SSA subscripts are simply allowed to propagate into Fulcra. Hence, Full SSA creates a larger number of variable names than any of the available merging policies. This means that the analysis results of Full SSA will be the closest to an actual flow-sensitive analysis of any code, but the analysis time is expected to suffer greatly as a penalty for increased solution precision.

3.2.2 Loop merge

The Loop Merge policy is the second most aggressive policy in terms of preserving program flow. This policy attempts to elimate flow that contributes assignment cycles to Fulcra's constraint graphs. The most common instance of a constraint graph cycle occurs across the back-edge of a loop. In an SSA representation, whenever a loop contains a variable with a value live-in to the header of a loop and a definition lying inside the loop, there will always be a merge node (μ) in the loop's header. This merge node accounts for the fact that definitions reach the loop header from both before and within the loop. In order for an assignment cycle to occur in SSA variable versions, a merge definition must be assigned into a string of subsequent versions and eventually flow back into the original merge node as an operand. In other words, there must be a path in the control flow graph such that the definition may propagate itself into two different operands of a merge node. In a reducible control flow graph, the only opportunity for such a situation occurs with variable definitions live-in across the back-edge of a loop and some other predecessor to the loop header. Therefore, the existence of a μ-node suggests the possibility that SSA will generate an assignment cycle.

The policy begins with the μ-node definition at the header of each loop. It then traverses all the definitions in the function and merges any that lie within the same loop as the μ definition. In the presence of nested loops, only a single definition name is alloted to the outermost loop

```
1: procedure LoopMerge(DefinitionList dlist)
2:     for all μ-def ∈ dlist do
3:         for all def ∈ dlist do
4:             if LoopContainsDef (LOOP(μ-def), def) then
5:                 MergeSubscripts (μ-def, def);
6:             end if
7:         end for
8:     end for
9: end procedure
```

Figure 3.2 Pseudocode for the Loop Merge flow policy algorithm.

because nested loop-bodies are treated as though they lie within the outermost loop. Therefore, all nested loops will share this definition name with the outermost loop. By merging all definitions contained within a loop, cycles are not added to the constraint graph simply by imposing flow on the program. Figure 3.2 shows pseudocode for the Loop Merge algorithm. The procedure called on line 4, *LoopContainsDef*, takes a loop and a definition as arguments and returns a boolean value indicating whether the definition lies within the loop.

This merging policy is intended to run very quickly during a processing phase before Fulcra sees the code. Therefore, the full constraint graphs are not available for cycle detection at the time when merging occurs. If full cycle detection were performed on the SSA, then this policy would offer no compute-time advantage over allowing the cycles to propagate into Fulcra and allowing it to perform cycle detection.

Figure 3.3 borrows the partial flow sensitive example from Figure 2.8 as a demonstration of the Loop Merge policy's capability to eliminate assignment cycles from the constraint graph. As the algorithm is run, (p_2, p_3, p_4) are all found to be defined within the same loop as the μ-node in block (2). These three variable versions from Figure 3.3(a) are merged into a single

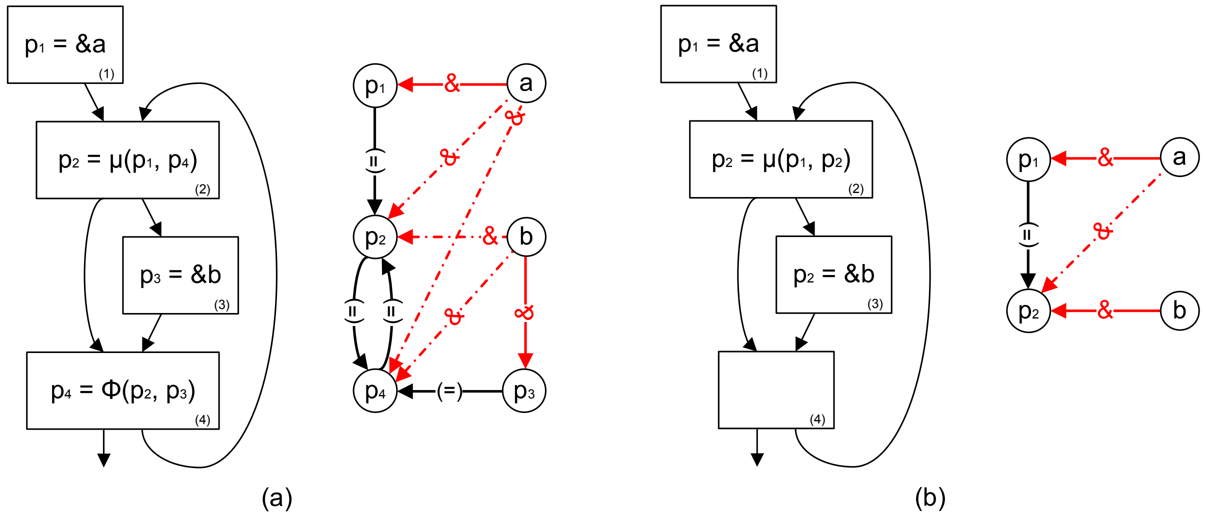

Figure 3.3 Loop Merge example. (a) SSA loop example borrowed from Figure 3.3. (b) Loop example after the Loop Merge flow policy has been applied.

variable name and become p_2 in Figure 3.3(b). This operation eliminates the the assignment cycle from Figure 3.3(a) since it is comprised of nodes p_2 and p_4 which become a single node after the loop merge algorithm is run. Additionally, the ϕ-node in block (4) is effectively removed because the version it defines and all its operands belong to the set of versions that were merged into p_2.

This policy is heuristic in nature, and therefore, many merge situations that may be beneficial to the quality of the analysis result. Imagine if the example from Figure 2.7 were present within the body of a loop. In that case, the Loop Merge policy would happily combine all the versions of p into one subscripted variable name and d's value would once again appear as though it flows into both a and b. Another situation when this heuristic makes a poor decision is in the presence of sequential definitions of a variable inside a loop body (Figure 2.5). Again,

```
procedure PhiMerge(DefinitionList dlist)
    for all φ-def ∈ dlist do
        PhiMergeRec (φ-def);
    end for
end procedure
procedure PhiMergeRec(Definition def)
    VISITED(def) ← 1;
    for all var ∈ RHS(def) do
        if VDCL(var) = VDCL(LHS(def)) then
            if VISITED(DEF(var)) = 0 then
                MergeSubscripts (def, DEF(var));
                PhiMergeRec (DEF(var));
            end if
        end if
    end for
end procedure
```

Figure 3.4 Pseudocode for the Disjoint Lifetime flow policy algorithm.

if such a situation occurs within a loop body, the Loop Merge policy will eradicate beneficial flow in favor of avoiding the creation of assignment edge cycles.

3.2.3 Disjoint lifetime

The final flow policy is named Disjoint Lifetime and is the most conservative policy in terms of program flow. In other words, the Disjoint Lifetime policy performs more merging of definition names than any other flow policy because the only effective renaming it preserves is that of disjoint variable live ranges. This merge policy attempts to combat Fulcra's run-time degradation by introducing only a minimal amount of flow to the pointer analysis framework.

Figure 3.4 provides a high level overview of the Disjoint Lifetime flow policy algorithm. On the top level, it traverses all the ϕ-nodes in a procedure and merges the ϕs' operands with their definition names. The algorithm then recurses on all the operands of that ϕ-node, visiting their definitions. If an instance of the current variable is found on the *right-hand side* (RHS) of the

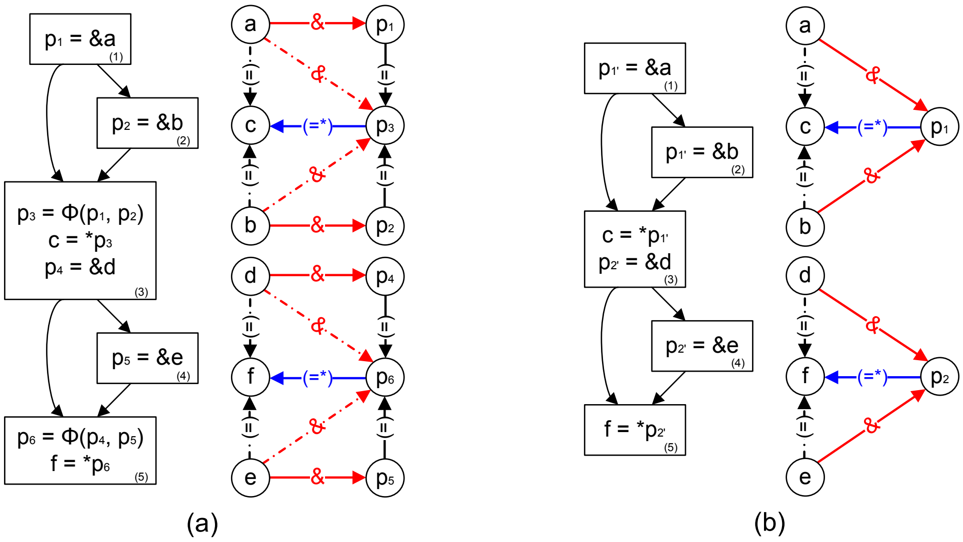

Figure 3.5 Disjoint Lifetime flow policy example. (a) Example of sequential code with two control merge points. (b) Disjoint Lifetime flow policy merging is applied. Variable versions p_1, p_2, p_3 become $p_{1'}$ and p_4, p_5, p_6 become $p_{2'}$. Both ϕ-nodes from blocks (3) and (5) are also eliminated by merging operations.

definition, the *left-hand side* (LHS) and RHS versions are merged together, and the algorithm recurses again on the RHS variable version. After this policy processes an entire procedure, all the inputs to every ϕ-node share a single definition name, which leaves these ϕ-nodes with a trivial function. Therefore, ϕ-nodes are functionally eliminated from the procedure after this policy is applied.

The effect of performing the Disjoint Lifetime flow policy is to split variable names that the programmer could have split manually. Essentially, any time all control flow converges upon a particular program point and a variable is redefined, that redefinition is given a new version name. Figure 3.5 depicts a situation that benefits from the Disjoint Lifetime policy. The code sequence shows two sequential control flow branches with a join point between them. Figure 3.5(a) shows the effect of full SSA renaming on this situation. The constraint graph is disconnected because of the definition of p_4 in block (3). Each subgraph in the constraint graph contains a pair of derived address edges as well as a pair of derived assignment edges. Figure 3.5(b) is an application of the Disjoint Lifetime policy over Figure 3.5(a). Notice that constraint graph is still divided into two subgraphs. The main difference between Figure 3.5(b) and Figure 3.5(a) is the number of p nodes, which is reduced from six down to two. A side effect of this reduction in node count is shown in the derivation edges. In Figure 3.5(b), the derivation process is simplified because four of the derived address edges found in Figure 3.5(a) are no longer necessary. The overall pointer relations are preserved across Figure 3.5(a) and (b), so Figure 3.5(b) offers less complexity at comparable precision.

As with the Loop Merge policy, the Disjoint Lifetime flow policy may merge away flow information that is beneficial to the analysis result. If the situation depicted in Figure 2.7 ever occurs within a procedure, the Disjoint Lifetime policy will certainly merge the beneficially subscripted versions of p into a single subscript. However, instances of sequential redefinitions (Figure 2.5) will not be merged by this flow policy because it relies on the presence of a control flow merge point to dictate the candidates for subscript merging.

3.3 Fulcra Modifications

The most noticeable change to Fulcra is the flow of intermediate representations. Although SSA subscripts are accessible, the complete SSA representation is not available from the AST. Specifically, ϕ-nodes are annotated to the CFG, but not the AST. Without ϕ-nodes, the various variable versions cannot be related to each other properly in Fulcra's internal constraint graph

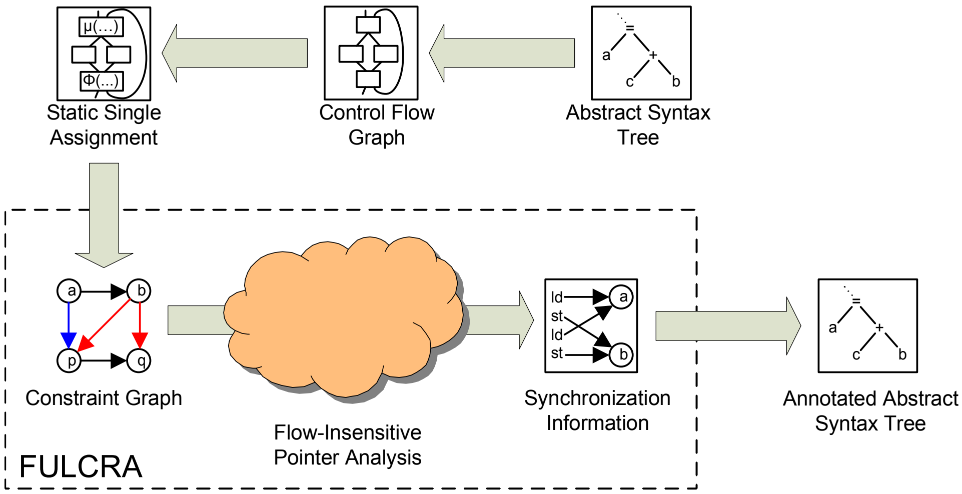

Figure 3.6 Revised Fulcra intermediate representation path. This new path allows for conversion of the AST to a CFG. The CFG is then transformed by SSA and fed into Fulcra.

representation. Section 2.3.2 stated that Fulcra derives initial constraint graph relations directly from the AST. This means the front end of Fulcra was modified to accept the CFG instead of the AST. The revised path of representations is shown in Figure 3.6. Due to the separation between Fulcra's internal constraint graph representation and IMPACT's AST, the conversion process was rather clean. In fact, Fulcra is constructed such that any given compiler could hook up its own AST and only modify a small front-end section of Fulcra's code.

Once Fulcra was able to read from the SSA-enhanced CFG, a few more internal code changes were still necessary. The largest internal modification necessary to support SSA in the context of partial flow-sensitivity was variable versioning. Fulcra already contained support for multiple versions of variables within its symbol-table and constraint graph. This was because its simple constraint graph grammar (see Figure 2.3) required the use of temporaries to model complex assignment behavior (such as $**p$) and the replication of heap object nodes necessary for heap specialization. Although support for variable versioning existed, the general policy was to assume a version of 1 on any object that was not a temporary or part of the heap. Several procedure declarations had to be modified to pass SSA subscripts from the CFG, into the constraint graph construction routines as well as the constraint graph node creation procedure itself.

During the constraint graph construction phase, Fulcra conditions the constraint graphs for the interprocedural analysis it will eventually perform. This requires the addition of standard interface nodes at function boundaries, in other words, parameters and return values from callee procedures. Both of these nodes were assumed to be version 1. Parameters were simple because SSA versioning automatically assigned version 1 to any use of a parameter not defined within the procedure. Return variables, on the other hand, could potentially hold a version. Since the addition of interface nodes occurred after the initial constraint graph constrain phase, the expressions carrying version numbers for return variables were no longer available. The answer was to store return variable version in the call-site data structure and refer back to it when the interface node was added for a particular callee.

The final front-end modification to Fulcra involved the modeling of ϕ-node behavior in a constraint graph. As stated in Section 2.1.1, a ϕ-node performs an operation which collects the definitions live at a control merge point into a new, single definition. In terms of flow-insensitive pointer analysis, this operation can be represented by a collection of assignment edges leading from each operand of the ϕ-node to the node defined by the ϕ-node assignment.

In other words, the LHS of the assignment receives the values of all the ϕ-node's operands from the RHS.

Once the above modifications were performed, Fulcra was fully capable of translating an SSA-enhanced CFG into its internal constraint graph notation and performing full-featured pointer analysis. However, another versioning issue arose after analysis completed and annotation of alias information occurred. In the back-end, Fulcra traversed the AST searching for expressions that would eventually become loads and stores. When it encountered such an expression, it would perform a query on the object being loaded from or stored to. Once again, this query process assumed a version of 1 for the general case and had to be modified. Fortunately, the SSA versioning information was stored to the variables in the AST during the SSA computation process, so the annotation process was easily corrected in the presence of partial flow-sensitivity.

CHAPTER 4

EXPERIMENTAL RESULTS

In this chapter, various aspects of partial flow-sensitivity are tested. These tests yield results that may be considered in a complete evaluation of the addition of partial flow-sensitivity to the Fulcra pointer analysis framework. The baseline of comparison for all analysis measurements is Fulcra's default configuration, which is its most accurate barring the addition of partial flow-sensitivity. This means Fulcra will perform a context-sensitive, heap-specialized, field-sensitive, flow-insensitive analysis upon the benchmarks listed. This baseline configuration corresponds to the *none* flow policy and is denoted in this way in all results figures. The other flow policies presented in the various results figures correspond to those covered in Section 3.2 (refer to Table 3.1). The following chapter evaluates five of the key characteristics of partial flow sensitivity; local variable coverage of SSA, the effects SSA has upon the number of subscripted variable names, the quality of pointer analysis results in the presence of partial flow sensitivity and the analysis time consequences of partial flow sensitivity.

4.1 SSA Coverage

SSA is the heart of partial flow sensitivity because the renaming it performs serves to isolate assignment models from each other when they occur in different parts of the control flow graph. However, not every variable is eligible for SSA computation. The limitation of the SSA

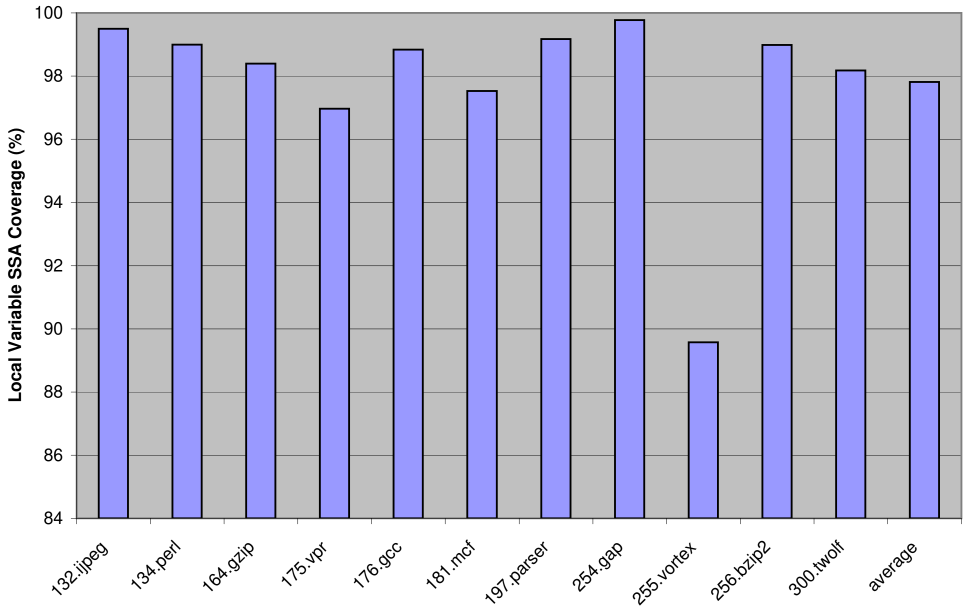

Figure 4.1 SSA coverage of local variables. Measured during SSA computation.

implementation discussed in Section 3.1.1 indicates that some local variables will be left out of the SSA representation. Obviously, with fewer variables available for SSA renaming, there are reduced opportunities for partial flow sensitivity to contribute meaningful flow and thereby simplify analysis and results. This arouses concerns for the coverage, because intuitively it is at least loosely correlated to analysis results.

The SSA coverage results for local variables in this implementation are shown in Figure 4.1. Given that SSA is intended for use with only local variables, global variables are not considered in these coverage numbers. These numbers are measured during the computation of SSA. That is, every variable that ends up represented by SSA is tabulated once and compared to the total number of local variables in the procedure.

Fortunately, the coverage is remarkably high in spite of the omission of address-taken local variables. In fact 8 of the 11 benchmarks represented here exhibit a coverage greater than 98% of local variables represented in the SSA graph. On the other hand, *255.vortex* is the benchmark with the lowest coverage, close to 90%. Evidently, a proportionally larger number of local variables have their addresses taken in *255.vortex* compared to the other benchmarks. At the same time, 90% coverage implies that a strong majority of the targeted variables will still have SSA representations and, hopefully, will allow partial flow sensitivity to contribute meaningful flow.

4.2 Variable Name Count Expansion

Variable name count expansion is inherent in this renaming process. For every new variable name introduced by SSA renaming, a corresponding node is added to the constraint graph along with the edges required to model the behavior of this subvariable. If the number of variable names increases too drastically, there is a distinct possibility that the added size and complexity imposed upon the constraint graph will impair execution to a degree that does not warrant the increased precision afforded by partial flow sensitivity.

The measurement of name count expansion due to SSA renaming is collected from the IMPACT SSA library after computation of the representation is concluded. This includes running the appropriate merging policy. Figure 4.2 offers a comparison of the number of variable versions created by SSA to the number of local variables processed by SSA (i.e., local variables that never have their address taken).

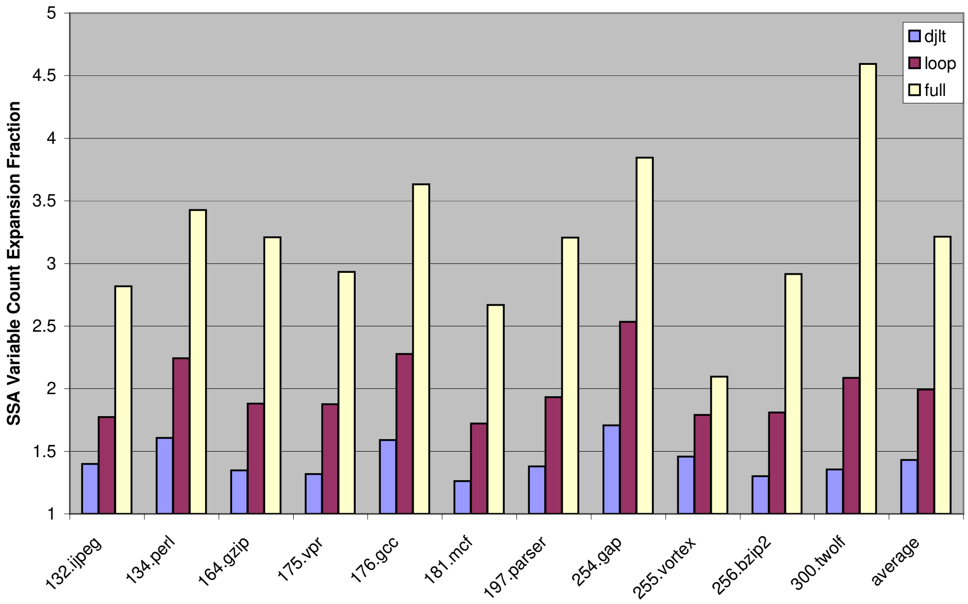

Figure 4.2 Local variable name count expansion due to the three flow sensitivity policies. These results are collected during SSA computation and normalized to the number of local variables found in each benchmark.

The subscript merging policies perform exactly as expected relative to the flow-insensitive and full SSA trials of each benchmark. The greatest expansion occurs as a result of the *full* SSA flow policy, *loop* contributes slightly less variable name expansion, and *djlt* generates the fewest number of subscripts. The steps down in expansion are fairly uniform, maintaining an approximate ratio of $3{:}2{:}1.5$ across all benchmarks. Outliers include *255.vortex*, with its flatter and smaller than typical expansions and *300.twolf*, because of the large expansion demonstrated by its *full* SSA configuration.

4.3 Solution Quality

This section of results presents some measurements of the analysis solution quality, the outcome of which is of core importance to the evaluation of partial flow sensitivity. However, it is unclear what metric to choose when evaluating an abstract concept as vague as *quality*. In the face of this ambiguity, Section 4.3.1 describes the methods devised to offer some measures of quality.

4.3.1 Counting methods

Partial flow sensitivity solution quality results are difficult to quantify. Other pointer analysis studies may be able to compare raw dependence data across trials if the tested features do not alter the number of objects in the analysis. A raw dependence count accumulates the number of access edges between static memory operations and constraint graph objects. This suffices as a valid comparison of pointer analysis result quality when the number of constraint graph objects remains constant. However, the nature of partial flow sensitivity is to split single objects into multiple versions dictated by the control flow of a program. This suggests that any partial flow sensitivity policy will contain more objects than a flow-insensitive analysis of the same code. In fact, varying the amount of flow yielded by the merging policies covered in Section 3.2 ensures object counts among the various merging policies will all differ from each other.

The replication of objects performed by partial flow sensitivity does not in and of itself make a results more complex. Rather, every split that occurs yields subobjects that are more

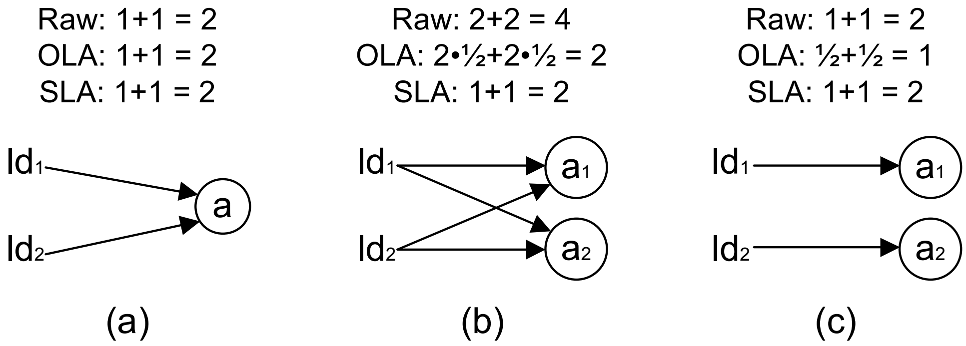

Figure 4.3 Counting methods example. (a) two static loads access a single object. (b) two static loads each access both versions of object a. (c) two static loads each access unique versions of object a.

specific in terms of liveness. The key hope is that these subobjects are less connected to load/store aliases than the parent object they were derived from. Figure 4.3 illustrates results gathered in different circumstances from three counting methods: raw count, object-level approximation of raw count (OLA), and symbol-level approximation of raw count (SLA). Before discussing the other counting methods, let us examine a problem with comparing a raw count of load/store accesses in Figure 4.3. In Figure 4.3(a), no objects have been split and analysis has concluded that ld_1 and ld_2 may load from the same object, a. In Figure 4.3(b), partial flow sensitivity is applied to a and has split it into two versions. However, the analysis reveals that both ld_1 and ld_2 may load from either version of a. In Figure 4.3(c), the same split of object a has occurred as in Figure 4.3(b), but the analysis reveals that ld_1 and ld_2 do not access a common version of a. A quick inspection of the number of edges in each situation reveals raw dependence numbers for Figure 4.3(a), Figure 4.3(b), and Figure 4.3(c) of 2, 4, and 2, respectively. However, the situation in Figure 4.3(b) is identical to that of Figure 4.3(a), because ld_1 and ld_2 may load from all versions of a. The raw count suggests the situation in Figure 4.3(b) is worse than Figure 4.3(a) by a factor of 2. In contrast, Figure 4.3(c) depicts an improvement over Figure 4.3(a) as ld_1 and ld_2 no longer alias. Again, the raw alias count suggests an inaccurate conclusion as the number of edges is the same, but splitting the node has made a meaningful contribution to the alias relation between ld_1 and ld_2.

The first option employed by this thesis to compensate for the replication of objects estimates a raw count by weighting each access edge by the inverse of the number of objects the target symbol is split into ($\frac{1}{num_{objects}}$). This method of counting estimates the raw count of access relations on the object level and therefore is named OLA. In terms of Figure 4.3(b), ld_1 has an access relation with objects a_1 and a_2. The OLA count is incremented by $\frac{1}{2}$ for each of ld_1's alias relations to the a objects because there are two versions of a present in the entire procedure. The OLA count for ld_2 is performed in the same way and added to the count from ld_1, making the final OLA count equal to 2.

OLA is an optimistic evaluation in some cases because it assumes that if a large number of versions exist for a single symbol, then the chance of two memory operations that alias with that symbol is inversely proportional to the number of object versions. In reality, the probability of memory operations aliasing with each other when they refer to the same symbol cannot be reliably modelled as a random function. For example, in Figure 4.3(c), suppose ld_2's alias relation was with a_1 instead of a_2. This situation is a markedly poorer result than the one shown in Figure 4.3(c) because ld_1 and ld_2 would now alias, but both cases yield the same OLA count.

A second means of evaluating the quality of pointer analysis results in the presence of varying object counts is to merge all subobjects back together as results are annotated to the source. This merging process preserves all the relations from each subobject and eliminates relations duplicated across multiple versions of a single object. The resulting graph is then comparable to a version in which no splitting of objects occurred and raw alias count information may be safely compared. The effect of merging all nodes at a symbol name level is to limit alias information to that level. It is because this counting method classifies relations on the symbol level that it is called SLA.

The use of SLA is, in some cases, more pessimistic than OLA. The basic difference being that OLA increments the raw count by a fraction every time an alias is linked to an object. If the worst case is observed, a memory operation aliases with all versions of an object, the total contribution to the OLA count for that memory operation is 1. If fewer than all versions of an object alias with a memory operation, the OLA contribution will be less than 1. SLA, on the other hand, cannot contribute fractions of a count. If a memory operation aliases with *any* version of an object, the SLA contribution of that memory operation is 1. A situation such as in Figure 4.3(c), although obviously a better result, is given the same SLA count as Figure 4.3(a) and (b). The bottom line is the only way for an SLA count to decrease across trials is by reducing the number of memory operations that link to symbol level aliases.

Situations that lower SLA count are those that completely eliminate basic pointer relations. An example of such a situation was given in Figure 2.7 when the use of control flow information eliminates the presence of a false dependence. There, an object d was derived to have its value placed in two other objects (a, b) in the flow-insensitive constraint graph from Figure 2.7(a). But when a notion of flow was imposed on the constraint graph in Figure 2.7(b), d's value is only derived to flow into one other object b. A false relation between two symbols was eliminated, which could in turn contribute to a lower SLA count.

4.3.2 Analysis quality results

Figures 4.4 and 4.5 are the results tabulated using the OLA counting scheme broken into load and store results respectively. The numbers graphed in these Figures have been normalized to the total raw alias counts from a trial with no partial flow sensitivity (i.e., the total height of the *none* bar). Aliases counts attributed to heap objects are on the top of the bars while stack and global objects are represented in the bottom bar division.

These benchmarks all react to the varying degrees of flow as expected. That is, merge policies that eliminate flow result in load/store alias counts closer to *none*. In general, these results suggest a direct correlation between the amount of flow imposed upon the pointer analysis framework and the reduction in OLA alias count. The *full* merging policy holds the best result in all benchmarks, which implies a 20% or greater reduction in OLA alias count for the typical case. These reductions in OLA count are also primarily attributed to reductions in aliases on the stack as the top segments of bars remain fairly constant across all merging policies for most benchmarks. One last general observation regarding OLA counts is they behave relatively similar across both load and store alias counts.

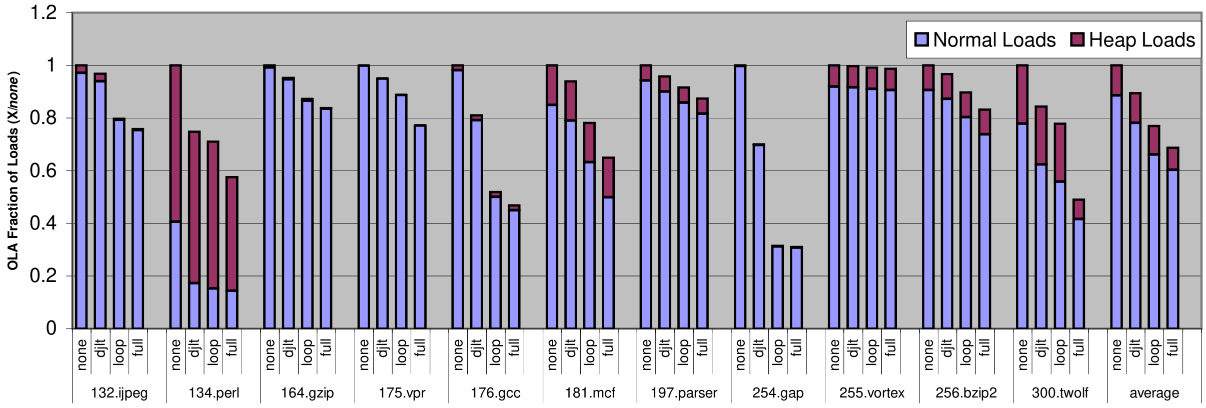

Figure 4.4 Analysis quality results for loads reported in OLA. These results are normalized to the total analysis result of loads with no flow sensitivity (*none*).

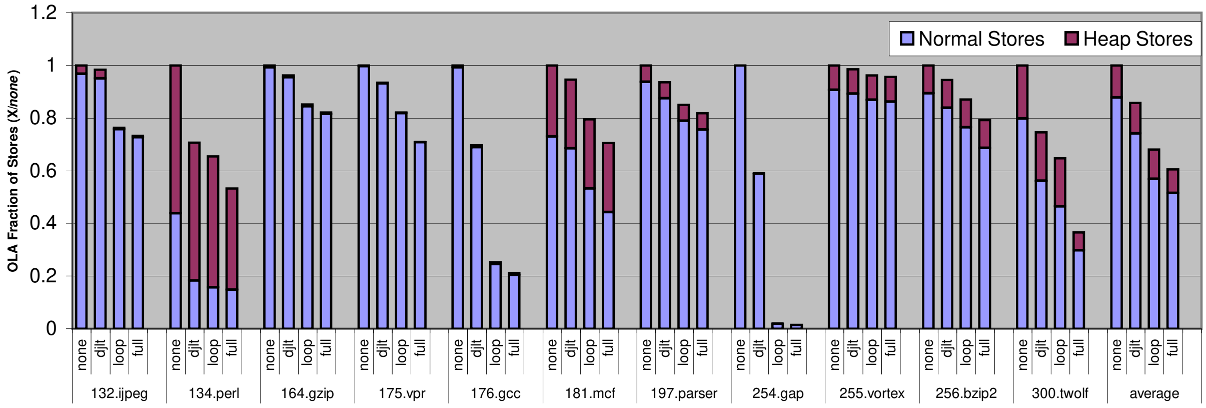

Figure 4.5 Analysis quality results for stores reported in OLA. These results are normalized to the total OLA count of stores with no flow sensitivity (*none*).

Figures 4.6 and 4.7 illustrate the SLA pointer analysis precision results for the various flow schemes. These numbers are collected from stack and global relations in each benchmark. The heap relations are not presented because merging the heap object versions yields results that are trivial in the face of partial flow sensitivity. The SLA counts of *djlt*, *loop*, and *full* are normalized to the SLA count of *none*, and the fraction is presented in the graph.

The general trends established by the OLA results appear once again in the more pessimistic SLA counts. That is to say the direct correlation between increasing amounts of flow and decreasing the SLA is once again evident by the behavior of the benchmarks. A more specific observation is the benchmarks exhibit generally the same reactions to flow given either counting method. For example, *255.vortex* demonstrates very flat behavior while *254.gap* flaunts a dramatic decrease in counts as flow approaches full SSA in Figures 4.4-4.7.

4.4 Analysis Compute-Time

The final class of results depict the impact partial flow sensitivity has on the compute-time of the pointer analysis information. The notion of compute-time considered here is simply wall-clock time as algorithmic derivations of worst-case run times can be highly pessimistic and would make comparisons amongst flow policies difficult to quantify. All timing measurements begin after the AST is loaded from disk and complete after Fulcra reaches a final analysis solution. Annotation of alias information back to the AST is not included in the timing span,

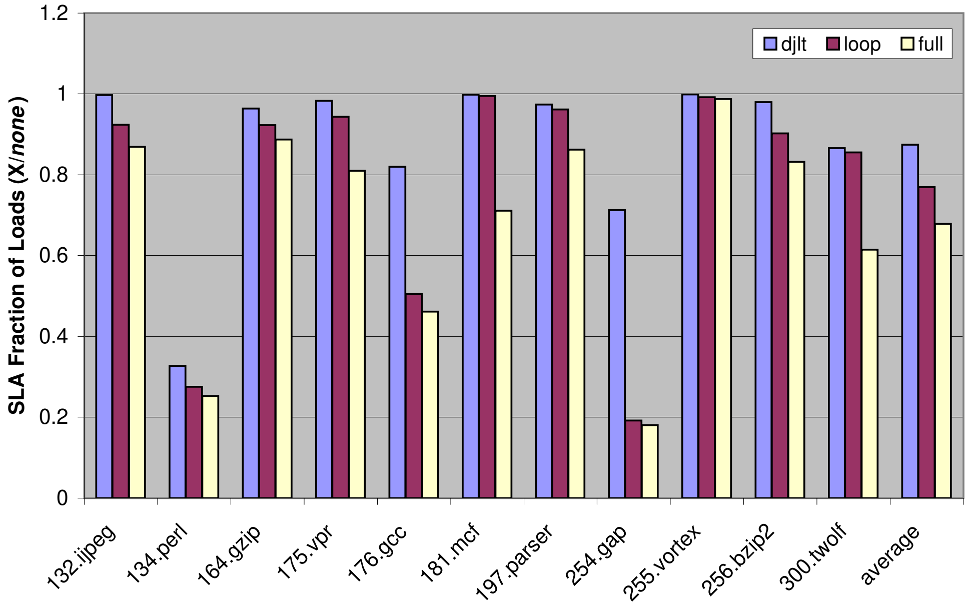

Figure 4.6 SLA analysis quality results for stack and local loads. These results are normalized to the load analysis result of the baseline.

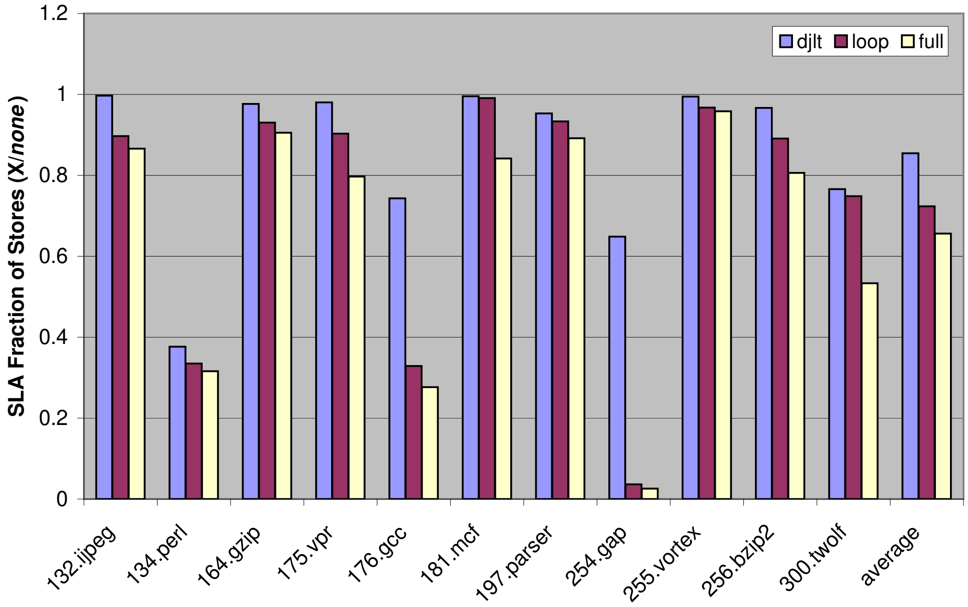

Figure 4.7 SLA analysis quality results for stack and local stores. These results are normalized to the store analysis result of the baseline.

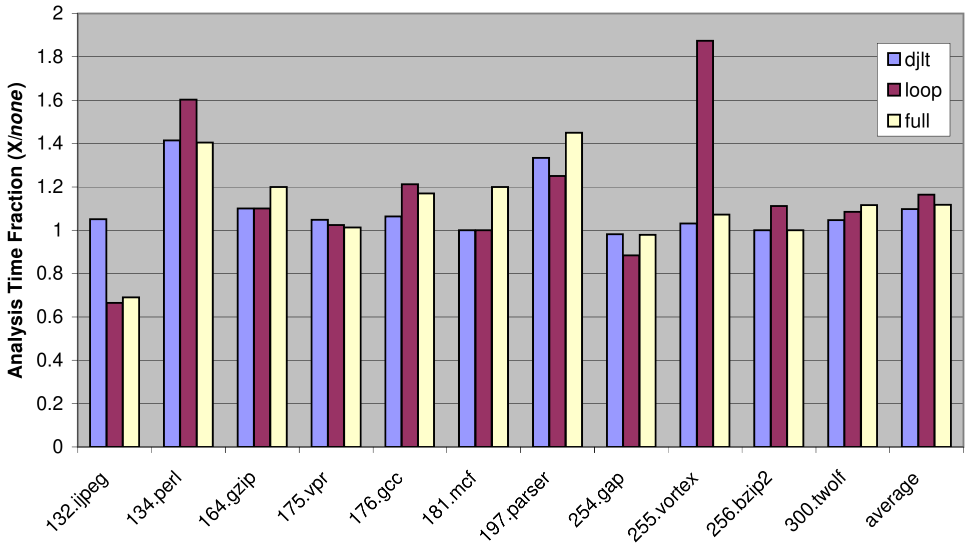

Figure 4.8 Analysis time results. These times are normalized to the time measurement of the baseline configuration.

but the analysis results are in a final form when the clock stops. This means that all the computations proposed in this thesis, as well as the analysis they feed, are included by the timing measurements.

The empirical timing measurements presented in Figure 4.8 are normalized to the baseline timing measurement. One important observation from this Figure is the fact that the timing penalties for all the flow policies are limited. No trial exceeds a timing fraction of 1.9 and most do not exceed 1.2. The timing fractions seem fairly irregular which suggests the presence of many complex correlations.

The *300.twolf* shows that more flow can lead to greater analysis time. This correlation is reinforced by the fact that *300.twolf* had the greatest degree of variable name count expansion from SSA. On the other hand, *132.ijpeg* and *254.gap* both have timing fractions below 1.

Hence, partial flow sensitivity actually decreases the amount of time these benchmarks took to run in spite of the added overhead to compute SSA, perform merging policies and increased node count introduced into the initial constraint graph. Essentially, partial flow sensitivity preconditions the code in a way that reduces the number of relations by specifying highly specialized pieces of objects. Sometimes this simplification of relations is enough to over compensate for the compute-time overhead it implies.

The spike in the graph on the *255.vortex loop* trial is actually due to SSA subscript merging policy overhead. The reason this timing fraction is so pronounced for this benchmark is due to the analysis time being very fast. The *255.vortex* is a relatively large program with fairly uninteresting pointer behavior, so the time measurement is slightly dominated by SSA construction and subscript merging in this particular case. The Loop Merge policy takes much longer than Disjoint Lifetime Merge in this case because the run-time is $O(n_\phi \times n_{def})$ as opposed to $O(n_{def})$ for Disjoint Lifetime Merge.

One final contributor to the slightly random nature of this graph is the influence of Fulcra's internal merging. After constraint graphs have been formed, Fulcra performs a pass over them and tries to identify pairs of nodes that will contribute identical results to the solution. When such a pair is encountered, they are merged into a single representative node. The solution quality is not affected by these merging operations, rather it is simply performed as an analysis compute-time consideration. When SSA preconditions the code and splits nodes apart, new node relations are formed. These new relations have been found to impair Fulcra's ability to detect equivalence of nodes it otherwise would in a flow insensitive mode.

4.5 Interpretation

Overall, the count reductions of OLA and SLA, for *full* SSA in particular, and the lack of any major analysis time penalty suggest that partial flow-sensitivity with full SSA renaming is an attractive option. Certain benchmarks, such as *134.perl*, *176.gcc*, and *254.gap*, gain a great deal of the stack/global object count reductions partial flow sensitivity has to offer with the Loop Merge policy. Based on this observation alone, the Loop Merge policy seems like a viable option to preserve scalability. However, examination of the analysis times in Figure 4.8 reveals that *134.perl* and *176.gcc* enjoy a healthy decrease in analysis time when run under the Full SSA flow policy. The *254.gap* suffers an approximate 8% cost in analysis time, but enjoys a 2% quality advantage according to the SAL counts. The *134.perl* also receives a healthy decrease in OLA count with regards to heap objects moving from Loop Merge to Full SSA, which is why examination of both quality count results is an important part of forming a conclusion. The rest of the benchmarks tend to follow a direct flow/quality relation and trade little scalability for analysis result quality. Although *255.vortex* demonstrates unimpressive reductions in OLA and SLA counts, its analysis time is nearly unaffected by the addition of full partial flow sensitivity. All in all, the results show partial flow-sensitivity with full SSA to be a beneficial addition to any pointer analysis framework striving for the highest quality results in the quickest time.

CHAPTER 5

FUTURE WORK

Figure 2.7 captures a situation in which one input to a ϕ-node offers no benefit to the quality of the derived constraint graph while another input does. The difference between the ϕ-node operands is that one is used in an expression before the control flow merge occurs and the other is not. This suggests the possibility of an intelligent merging policy that distinguishes between cases similar to the example in Figure 2.7.

The implementation of such a merging policy would traverse all the ϕ-nodes in a procedure and examine their operands. If an operand is found with no ordinary uses (uses other than as a ϕ-node operand), it is merged with the version defined by the ϕ-node. The conditions for merging in this case imply that any variable version with an ordinary use will maintain its original version, therefore no information is eliminated from that constraint graph that may contribute to the quality of the results. Because the algorithm for this flow policy is exactly the same as the Disjoint Lifetime policy with an extra to be satisfied before merging, the variable name count expansion will be lower bounded by that of the Disjoint Lifetime and upper bounded by the full SSA flow policy's expansion. If, in fact, no quality-modifying information is removed in this merging policy, then the expected behavior would be quality counts identical to full SSA and potentially a reduction in analysis time.

CHAPTER 6

CONCLUSION

This thesis evaluated the notion of partial flow-sensitivity by empirical means. It also proposed two novel flow-sensitivity policies for the purpose of further characterizing the effects of flow on an otherwise flow-insensitive pointer analysis framework. Although the primary objective was to improve the quality of pointer analysis, a secondary goal was to do so at little cost to scalability. As such, the design and characterization of flow policies served to establish multiple data points to correlate analysis quality and scalability. In the end, the original notion of partial flow-sensitivity is shown to provide substantial improvements in the quality of pointer analysis results at a minimal cost to practical scalability.

REFERENCES

[1] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002, pp. 25–34.

[2] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994.

[3] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996, pp. 32–41.

[4] E. M. Nystrom, H. Kim, and W. W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer," in *The Proceedings of the 11th Static Analysis Symposium*, August 2004.

[5] A. Milanova, A. Routev, and B. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," in *Proceedings of International Symposium on Software Testing and Analysis*, 2002, pp. 1–11.

[6] E. M. Nystrom, H. Kim, and W. W. Hwu, "Importance of heap specialization in pointer analysis," in *The Proceedings of Program Analysis for Software Tools and Engineering*, June 2004.

[7] E. M. Nystrom, "Fulcra pointer analysis framework," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[8] R. P. Wilson, "Efficient context-sensitive pointer analysis for C programs," Ph.D. dissertation, Stanford University, Stanford, CA, 1997.

[9] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1999, pp. 133–146.

[10] S. Zhang, B. G. Ryder, and W. Landi, "Program decomposition for pointer aliasing: A step toward practical analyses," in *Proceedings of The European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996, pp. 81–92.

[11] S. Zhang, B. G. Ryder, and W. A. Landi, “Experiments with combined analysis for pointer aliasing,” in *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering*, June 1998.

[12] R. Hasti and S. Horwitz, “Using static single assignment form to improve flow-insensitive pointer analysis,” in *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation*, June 1998, pp. 97–105.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[14] J. Choi, R. Cytron, and J. Ferrante, “Automatic construction of sparse data flow evaluation graphs,” in *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991, pp. 55–66.

[15] R. Ballance, A. Maccabe, and K. Ottenstein, “The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages,” in *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation*, 1990, pp. 257–271.

[16] E. M. Nystrom, H. Kim, and W. W. Hwu, “Scalable, precise context-sensitive top-down process for modular points-to analysis,” University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-03-03, March 2003.